# Flexibility-Based Traffic Flow Optimisation in Lifelong Multi-Agent Path Finding

Peiqian Lin
Monash University
Melbourne, Australia
lpq66632@gmail.com

Zhe Chen
Monash University
Melbourne, Australia
zhe.chen@monash.edu

David L. Dowe
Monash University
Melbourne, Australia
david.dowe@monash.edu

Daniel Harabor
Monash University
Melbourne, Australia
daniel.harabor@monash.edu

## ABSTRACT

Lifelong Multi-Agent Path Finding (LMAPF) is a coordination problem where a team of agents works continuously to complete tasks in a shared environment. Recent approaches for LMAPF combine fast myopic planners with high-level congestion-aware path guidance; they can coordinate thousands of agents in real-time. A main drawback is that congestion estimates assume each agent will follow one specific, pre-computed, time-independent path. This overlooks the existence of multiple cost-equivalent paths, leading to potentially inaccurate predictions and low-quality guidance. In this paper, we propose a novel method for computing congestion-aware guidance that reasons about this ambiguity. Instead of committing to a single route, our approach calculates a probabilistic traffic flow by considering the likelihood of travelling across all cost-equivalent paths. By reasoning over probability distributions to determine expected congestion we generate more robust and accurate guidance heuristics, which in turn enable planners to make more informed decisions. We conduct extensive experiments on large-scale LMAPF benchmarks with up to 16,000 agents. Results show that our approach consistently outperforms baseline algorithms in system throughput.

## KEYWORDS

Heuristic Search; Flexibility; Multi-Agent Path Finding; Lifelong MAPF; Combinatorial Optimisation

**ACM Reference Format:**
Peiqian Lin, Zhe Chen, David L. Dowe, and Daniel Harabor. 2026. Flexibility-Based Traffic Flow Optimisation in Lifelong Multi-Agent Path Finding. In *Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), Paphos, Cyprus, May 25 – 29, 2026*, IFAAMAS, 9 pages. https://doi.org/10.65109/IRBO1733

## 1 INTRODUCTION

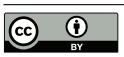
Many large-scale industrial applications, such automated warehouses and computer games, require coordinating hundreds or even thousands of simultaneous agents. The agents navigate a shared physical space and must complete a stream of continuously arriving tasks. The core problem at the heart of these applications is known as Lifelong Multi-Agent Path Finding (LMAPF).

A substantial body of research in this area focuses on computing optimal [13, 16, 18] or bounded-suboptimal solutions [1, 3]. These works provide strong solution quality guarantees but are generally considered too slow for the dynamic/online nature of LMAPF. Other works, such as Rolling-Horizon Collision Resolution (RHCR) [14], decompose the problem into a sequence of smaller, more manageable instances. Yet even these approaches struggle to manage teams with thousands of agents, which are required by modern industrial applications.

To achieve greater scalability recent work has suggested a new paradigm: using high-level "path guidance" to direct fast but myopic low-level planners. A particularly successful example is Traffic Flow Optimisation (TFO) [4]. Here the high-level planner computes a recommended trajectory that each agent tries to follow, in order to reach its destination efficiently. The guide path functions as a powerful heuristic for low-level PIBT [15], a simple but effective multi-agent coordination strategy. By making agents aware of potential bottlenecks, path guidance helps prevent the severe traffic that would otherwise emerge from agents greedily pursuing their individual shortest paths. Other similar works include Space Utilization Optimization (SUO) [8], where guidance is used as a tie-breaker between optimal paths, and GGO [23, 24] which learns a weighted "guidance graph" to help steer agents.

TFO computes congestion costs based on the strong assumption that each agent will follow one specific, pre-calculated, time-independent path to its destination. However, this assumption overlooks a common reality: for any given agent, there often exist multiple distinct, cost-equivalent paths. A low-level planner (such as PIBT) is free to pursue any of these optimal paths at runtime, and its choice may be influenced by the immediate, local movements of other agents. By committing to only one of these paths for congestion calculation, existing methods generate a brittle and potentially inaccurate picture of future traffic flow, leading to inefficient guidance.

In this paper, we address this limitation by proposing a novel method for generating congestion-aware guidance that explicitly reasons over this ambiguity. Instead of relying on a single-path assumption, our approach computes a data structure that represents all cost-equivalent paths and then calculates a probabilistic traffic flow based on the likelihood of travel across all of them. This





*Proc. of the 25th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2026), C. Amato, L. Dennis, V. Mascardi, J. Thangarajah (eds.), May 25 – 29, 2026, Paphos, Cyprus.* © 2026 International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). https://doi.org/10.65109/IRBO1733

probability-based congestion model allows us to generate more robust and accurate guidance heuristics, resulting in significant improvements in system throughput in large-scale LMAPF scenarios.

To evaluate our proposal we conduct extensive empirical evaluations on large-scale LMAPF benchmarks with up to 16,000 agents. Results demonstrate that our new approach consistently outperforms baseline algorithms. We also show that our method's single-shot congestion estimation is so effective that it surpasses the baseline approach even when the latter is given additional computation time for iterative refinement, highlighting the power of reasoning about cost-equivalent paths and the probabilistic traffic flow.

## 2 PROBLEM DEFINITION

The **Lifelong Multi-Agent Path Finding (LMAPF)** is the problem of coordinating a set of agents that continuously move and perform tasks in a shared environment [14, 20]. The problem consists of a directed graph $G = (V, E)$, which represents a 2D grid map where $V$ is the set of vertices (locations) and $E$ is the set of edges connecting adjacent vertices and a set of $n$ agents, $A = \{a_1, a_2, \ldots, a_n\}$.

Time is discretized into timesteps $t = 0, 1, 2, \ldots, T$, where $T$ is a finite time horizon. At each timestep, every agent $a_i \in A$ occupies a single vertex, and can perform an **action** to move to an adjacent vertex or to *wait* at its current vertex.

A **task** for an agent $a_i$ is to travel from its current location to a newly assigned goal location $g_i \in V$. A **plan** (or path) for an agent $a_i$ is a time-indexed sequence of vertices, $\pi_i = (\pi_i(0), \pi_i(1), \ldots, \pi_i(k))$, where $\pi_i(t)$ is the vertex occupied by agent $a_i$ at timestep $t$. A **solution** to an LMAPF instance is a set of evolving plans for all agents, $\{\pi_1, \pi_2, \ldots, \pi_n\}$, that remains valid over the entire duration $T$.

A solution is considered **valid** if it contains no conflicts. For any two agents $a_i, a_j \in A$ (where $i \neq j$), the following conflicts are forbidden:

- A **vertex conflict** occurs if both agents occupy the same vertex at the same timestep: $\pi_i(t) = \pi_j(t)$.
- An **edge conflict** (or swap conflict) occurs if the agents traverse the same edge in opposite directions during the same timestep: $\pi_i(t) = \pi_j(t+1)$ and $\pi_i(t+1) = \pi_j(t)$.

In the lifelong setting, once an agent $a_i$ reaches its goal $g_i$, it is immediately assigned a new task with a new goal location $g_i'$. We follow Li et al. [14] and assumes that there's an external component decides the incoming goal locations. The primary **objective** in LMAPF is to maximise the system's **throughput**, defined as the total number of tasks completed by all agents within the operational time limit $T$.

## 3 BACKGROUND

In this section, we review several prior works that form the basis of our contribution.

### 3.1 Priority Inheritance with Backtracking (PIBT)

PIBT is an algorithm combining rule-based and prioritized planning approaches [15]. It repetitively plans for each agent at every timestep based on the assigned priority until it is terminated. We describe PIBT in Algorithm 1.

**Algorithm 1** PIBT plans the next move for each agent at every timestep. For agent $a_i$, the current location is $\pi_i[t]$ and the next location is $\pi_i[t{+}1]$. When a conflict arises, PIBT resolves the conflict by using the priority number assigned to each agent. A priority number assigned to agent $a_i$ is denoted by $p_i$.

```
1:  p_i ← ε_i: for a_i ∈ A s.t. ε_i ∈ [0, 1) and ε_i ≠ ε_j (i ≠ j);
2:  for timestep t = 0, 1, 2, ... until terminates do
3:      for a_i ∈ A do π_i[t] = g_i ? p_i ← ε_i : p_i ← p_i + 1;
4:      sort A in decreasing order of p_i;
5:      for a_i ∈ A do
6:          if π_i[t + 1] = ⊥ then PIBT(a_i, ⊥);

7:  function PIBT(a_i, a_j)
8:      C ← Neigh(π_i[t]) ∪ {π_i[t]};
9:      sort u ∈ C in increasing order of dist(u, g_i);
10:     for v ∈ C do
11:         if ∃a_k ∈ A s.t. π_k[t + 1] = v then continue;
12:         if (a_j ≠ ⊥) ∧ (π_j[t] = v) then continue;
13:         π_i[t + 1] ← v;
14:         if ∃a_k ∈ A s.t. (π_k[t] = v) ∧ (π_k[t + 1] = ⊥) then
15:             if PIBT(a_k, a_i) is invalid then continue;
16:         return valid;
17:     π_i[t + 1] ← π_i[t].
18:     return invalid;
```

PIBT first assigns an priority to each agent (line 1). Then, at each timestep, it updates the priority of each agent and allows every agent who has not decided its next move to execute PIBT($a_i$, $\perp$), according to the priority order (lines 2-6).

When PIBT($a_i$, $\perp$) is called, the algorithm sorts all neighbour locations and its current location, $C$, of $a_i$ in increasing order of $dist(u, g_i)$, where $dist(u, g_i)$ represents the single agent shortest distance between the chosen location $u$ and $a_i$'s goal location $g_i$ (line 9). Subsequently, PIBT skips the current location if a vertex conflict (line 11) or a swap conflict (line 12) exists. If no conflict occurs, the algorithm reserves $v$ for $a_i$ and proceeds to line 14, which checks if there is any agent $a_k$ who has not determined its next move and is occupying $v$. If $a_k$ exists, PIBT($a_k$, $a_i$) will be called recursively (line 15), where agent $a_k$ tries to decide its next move by inheriting the priority from $a_i$. The algorithm skips the current location $v$ if PIBT($a_k$, $a_i$) returns invalid, meaning $a_k$ failed to move to a neighbour location. Otherwise, if $a_k$ does not exist or PIBT($a_k$, $a_i$) returns valid, $a_i$ occupies $v$ in the next timestep and returns valid. If $a_i$ fails to move to any locations in $C$, it returns invalid and stays at its current location at the next timestep. By incrementing the priority of each agent by 1 and resetting its priority when the agent reaches its goal, PIBT grantees that each agent will reach its goal location at least once if the $G$ is bi-connected [15].

The reachability guarantee on a bi-connected graph is extremely useful in the Lifelong MAPF problem, since unlike classic MAPF [20] agents are not required to stay at their goal locations forever. However, guided by the myopic individual shortest distance heuristic, the algorithm plans inefficient actions and often leads to severe traffic congestion in dense problems.

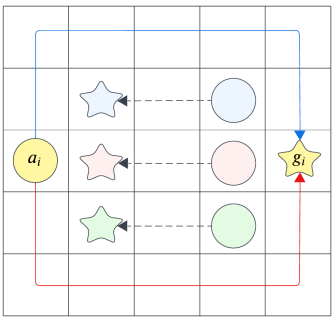



**Figure 1: An example of multiple congestion-aware optimal paths. Three agents (blue, pink, and green) contribute significant congestion in the centre of the grid map. Agent $a_i$ has two congestion-aware optimal paths, shown as the red and blue lines.**

## 3.2 Traffic Flow Optimisation

To overcome PIBT's myopic behaviour, Chen et al. [4] proposed to guide the myopic planner with high-level, congestion-aware routes. Their approach, inspired by the Traffic Assignment Problem (TAP), computes time-independent "guide paths" for each agent that proactively avoid likely traffic jams.

To achieve this, they first model traffic congestion. After an initial set of shortest paths is computed, they calculate two forms of congestion cost: **vertex congestion**, which penalises locations that many paths pass through, and **contraflow congestion**, which heavily penalises edges traversed by agents in opposite directions. These costs are then used to update the graph's edge weights, making congested routes more expensive to traverse. The system then iteratively re-plans paths for agents using these updated costs, converging towards a state where traffic flow is more balanced.

The resulting congestion-aware paths are not executed directly. Instead, they serve as "guide paths" to generate a more intelligent **guide heuristic** for PIBT. At each timestep, instead of myopically choosing the move that minimizes the direct distance to its goal, an agent uses this heuristic to select the move that best follows its assigned guide path. By steering agents along these globally-aware routes, this method effectively mitigates congestion before it occurs, leading to substantial improvements in system throughput in large-scale scenarios.

Related work by Han and Yu [8] uses a heuristic to tie-break between optimal shortest paths, favouring less congested routes. In contrast, the more aggressive method from Chen et al. [4] finds potentially longer, detour paths by directly modifying edge costs based on anticipated traffic. This latter approach has been shown to be more effective for guiding myopic planners like PIBT.

# 4 METHODOLOGY

The traffic flow optimisation method from Chen et al. [4] computes a single, congestion-aware guide path for each agent. This high-level route is then translated into a **guide heuristic** that replaces the standard distance metric within the low-level PIBT planner, making each agent's decisions aware of anticipated traffic.

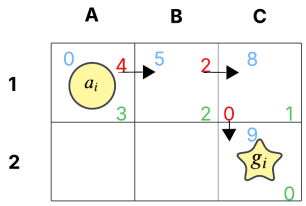



**Figure 2: An example of costs components. Blue numbers are $g(n)$, red numbers are $traf(n_1, n_2)$ when traversing edges, and green numbers are $h(n)$.**

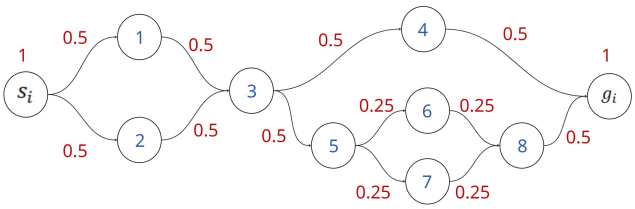



**Figure 3: An example of flow propagation along optimal paths. Blue numbers indicate node indices, and red numbers represent the flow to be updated on the edges between nodes.**

However, the method's core assumption is fragile because it commits to only one path, even when multiple **cost-equivalent optimal paths** exist. This constraint becomes problematic as the low-level planner may not strictly follow the prescribed route. PIBT's policy, for instance, allows lower-priority agents to yield and deviate in response to local interactions with higher-priority agents, meaning the actual path taken can differ from the single path used to model congestion.

By ignoring other equally cost paths, the high-level planner generates an incomplete and potentially inaccurate picture of future traffic flow, as agents can deviate from their assigned guide path at runtime. The existing approach [4] attempts to compensate for this flaw through frequent and computationally expensive re-planning. The success of the "anytime improvement" process described by Chen et al. [4] highlights this weakness, as its effectiveness comes not just from optimising a single route but, more critically, from allowing re-planning to introduce the **alternative path choices** that correct for the model's initial blind spots along the simulation and execution. Ultimately, by artificially constraining an agent to a single optimal path when others exist, we create a strategic handicap, similar to a principle in chess where having relatively more available moves is an advantage [11]. This can also be thought of in machine learning terms, such as minimum message length (MML) [21] or sharpness-aware minimisation, where an appropriately quantified broader peak is preferable [22][6, sec. 6][7, 10].

To address this issue, we propose a novel method for generating congestion-aware guidance that reasons over the entire set of cost-equivalent paths. Our approach is founded on a key assumption: when a low-level planner encounters multiple actions with identical cost-to-go heuristic values, it chooses among them with uniform probability.

**Algorithm 2** Compute OCPG

**Input:** start $s_i$, goal $g_i$, traffic flows $\mathcal{F}$
**Output:** OCPG $G'_i = (V'_i, E'_i)$
**Phase 1: Exhaustive A* Search**
Initialise OPEN, CLOSED, $g(s_i) \leftarrow 0$, $Parents(n) \leftarrow \emptyset$ for all $n$
OPEN.push($s_i$), $optimal\_cost \leftarrow \infty$
**while** OPEN is not empty **do**
  $n \leftarrow$ OPEN.pop()
  **if** $n = g_i$ and $optimal\_cost = \infty$ **then**
    $optimal\_cost \leftarrow f(g_i)$
  **if** $f(n) > optimal\_cost$ **then break**
  Add $n$ to CLOSED
  **for** $n' \in Neigh(n)$ **do**
    $new\_g \leftarrow g(n) + 1 + traf(n, n')$
    **if** $n'$ not in OPEN and $n'$ not in CLOSED **then**
      $g(n') \leftarrow new\_g$, $Parents(n') \leftarrow \{n\}$
      OPEN.push($n'$)
    **else if** $n'$ in OPEN and $new\_g < g(n')$ **then**
      $g(n') \leftarrow new\_g$, $Parents(n') \leftarrow \{n\}$
      OPEN.decrease_key($n'$)
    **else if** $new\_g = g(n')$ **then**
      $Parents(n') \leftarrow Parents(n') \cup \{n\}$

**Phase 2: OCPG Reconstruction**
$V'_i \leftarrow \{g_i\}$, $E'_i \leftarrow \emptyset$, queue $Q \leftarrow \{g_i\}$
**while** $Q$ is not empty **do**
  $child \leftarrow Q$.pop()
  **for** $parent \in Parents(child)$ **do**
    $E'_i \leftarrow E'_i \cup \{(parent, child)\}$
    **if** $parent$ not in $V'_i$ **then**
      $V'_i \leftarrow V'_i \cup \{parent\}$, $Q$.push($parent$)
**return** $G'_i = (V'_i, E'_i)$

**Algorithm 3** We add the traffic flow caused by $a_i$ to $\mathcal{F}$ in this algorithm.

**Input:** start $s_i$, goal $g_i$, OCPG $G'_i = (V'_i, E'_i)$
**Output:** A traffic flow update from $a_i$ on edges along Cost-equivalent optimal paths from $s_i$ to $g_i$
Initialise OPEN $\leftarrow \{s_i\}$, WEIGHT($s_i$) $\leftarrow 1$
**for** $n \in V'_i$ **do** READY($n$) $\leftarrow |Parents(n)|$
**while** OPEN $\neq \emptyset$ **do**
  node $u \leftarrow$ OPEN.pop_front()
  **if** READY($u$) $> 0$ **then**
    OPEN.push_back($u$)
  **else** ▷ READY($u$) $= 0$
    $curr_w \leftarrow$ WEIGHT($u$)$/|Children(u)|$
    **for** $v \in Children(u)$ **do**
      READY($v$) $\leftarrow$ READY($v$) $- 1$
      **if** $v \notin$ OPEN **then**
        WEIGHT($v$) $\leftarrow curr_w$
        **if** READY($v$) $= 0$ **then**
          OPEN.push_front($v$)
        **else**
          OPEN.push_back($v$)
      **else** ▷ $v \in$ OPEN
        WEIGHT($v$) $\leftarrow$ WEIGHT($v$) $+ curr_w$
      $f^i_{u,v} \leftarrow f^i_{u,v} + curr_w$

Building on this, our method first computes a data structure representing all optimal paths for an agent (Algorithm 2). From this, we derive the probability $p^i_{u,v}$ that the agent $i$ will traverse any given edge $(u, v) \in E$ and $p^i_v$ for any given vertex $v \in V$ (Algorithm 3). By aggregating these probabilities from multiple agents, we define **traffic flow**, captured by two metrics: $f_{u,v}$ for the flow on each directed edge, and $f_v$ for the flow through each vertex. We define $\mathcal{F}$ as the set of all traffic flows on each edge and each vertex. This traffic flow model allows us to estimate and optimise congestion costs, where higher flow on an edge results in a higher traversal cost. These costs are then used to compute a final, congestion-aware, cost-to-go heuristic for each agent. This new heuristic replaces the simple shortest-distance metric $dist(u, g_i)$ in planners like PIBT (Algorithm 1, line 9), guiding agents with the understanding of system-wide congestion.

## 4.1 Identifying All Cost-Equivalent Optimal Paths

Our search for optimal paths uses a standard A* evaluation function, $f(n) = g(n) + h(n)$, where the cost function is adapted to be congestion-aware. The term $g(n)$ represents the cumulative cost from the start node $s_i$ to the current node $n$, incorporating both the free-flow movement cost, which is 1 for each step, and an additional traffic cost $traf(u, v)$, computed with the given current set of background traffic flow $\mathcal{F}$. In this work, we follow Chen et al. [4] and define the traffic cost, $traf(u, v)$, for moving from $u$ to $v$ as

$$traf(u, v) = \lfloor (f_{u,v} + 1) \times f_{v,u} + \frac{f_v}{2} \rfloor \quad (1)$$

Here, the first term models the cost of *contraflow congestion*, while the second term models the expected delay from *vertex congestion*. We take the floor of the result to ensure integer-only costs, as the concept of multiple strictly cost-equivalent paths becomes ill-defined with floating-point costs. This can be seen as a form of approximation of cost equivalence. The heuristic, $h(n)$, is the unweighted shortest distance cost from $n$ to the goal $g_i$. As illustrated in Figure 2, traversing the edge $(A1, B1)$ adds a movement cost of 1 and a traffic cost of 4 to the initial $g(\text{A1}) = 0$, resulting in $g(\text{B1}) = 5$. Because $h(n)$ does not include traffic costs, it remains admissible, guaranteeing that the path found is optimal with respect to our congestion-aware cost function.

Now, we explain Algorithm 2, a modified A* procedure that identifies all cost-equivalent optimal paths. Unlike a classic A* that terminates upon reaching the goal, our algorithm continues expanding nodes until the $f(n)$ of the best node in the OPEN list exceeds the cost of the optimal path. This exhaustive expansion is conceptually similar to the construction of a Multi-Decision Diagram (MDD) [17].

However, a critical distinction exists. An MDD is a time-layered data structure that records all paths of a *fixed, predetermined temporal length*. In contrast, our method derives an **Optimal-Cost Path**

**Graph (OCPG)** that captures all paths sharing the same minimal *congestion-aware cost*, regardless of their length in timesteps. This is crucial because a shorter, more congested route may have the same total cost as a longer, less-congested one.

Formally, for an agent $a_i$ with start $s_i$ and goal $g_i$, its OCPG, denoted $G'_i = (V'_i, E'_i)$, is a directed subgraph of the input graph $G = (V, E)$. The vertex set $V'_i \subseteq V$ contains every vertex that lies on at least one cost-equivalent optimal path from $s_i$ to $g_i$. Similarly, the edge set $E'_i \subseteq E$ contains every directed edge $(u, v)$ that constitutes a step on at least one of these optimal paths. The resulting OCPG is a Directed Acyclic Graph (DAG) that concisely represents the entire space of optimal choices for the agent. This distinction is crucial, as it allows us to capture a more diverse set of strategically different, yet equally optimal, routes for our probabilistic model.

Algorithm 2 consists of two main phases. The first phase (lines 1-18) is an exhaustive A* search that discovers all nodes and edges belonging to any cost-equivalent optimal path. It begins like a standard A* search, but when it first reaches the goal, it records the optimal path cost, *optimal_cost*, and continues its expansion. The search only terminates once the smallest $f$-value in the *OPEN* list exceeds *optimal_cost*. During this process, when an equally optimal path to a node $n'$ is found, the new predecessor is added to a list of parents, $Parents(n')$, instead of being discarded. The second phase (lines 19-26) reconstructs the OCPG from these backpointers. It begins a backward traversal from the goal node $g_i$ , using the Parents map to navigate back to the start $s_i$ and build the forward-pointing edges of the OCPG. Figure 3 shows an example of OCPG.

## 4.2 Probabilistic Traffic Flow

Once the OCPG is constructed, we calculate the **probabilistic traffic flow** by propagating one unit of flow from the start node $s_i$. The flow calculation follows two rules based on the OCPG's topology:

**Flow Splitting:** The total flow arriving at a node $u$ is divided equally among all its outgoing edges. This reflects an equal probability of choosing any optimal next step.

**Flow Merging:** The total flow arriving at a node $v$ is the sum of the flows from all its incoming edges. This aggregates the probabilities from all optimal paths that converge on that node.

This propagation continues until the flow reaches the goal. These resulting flow values represent the probability that an agent, choosing randomly among equivalent optimal actions, will use that specific edge or vertex. Aggregating all flows from each agent's OCPG on each directed edge $(u, v)$ defines the edge's probabilistic traffic flow, $f_{u,v}$ , while the total flow arriving at each vertex $v$ defines the vertex flow, $f_v$. Figure 3 demonstrates how flows are distributed among an OCPG.

Here, we introduce Algorithm 3, which computes the probabilistic traffic flow contribution of any agent $a_i$. We begin by assigning one unit of flow to $s_i$. For each node $n$ in $V'_i$ (from Algorithm 2), we initialise READY($n$) as the number of its parents. The READY($n$) ensures that all incoming flow to $n$ has completely been merged before the node propagates its flow. In particular, READY($n$) = 0 indicates that $n$ is **ready**, while a positive value means **not ready**.

In each iteration, we pop a node $u$ from the front of OPEN (line 4), and then check if it is ready. If $u$ is not ready, we push it to the back of OPEN and continue with the next node. If it is ready, we start the propagation. We compute the flow that $u$ will pass to its child or children, splitting it evenly among them (line 8).

For each child $v$ of $u$, we decrement the READY($v$) by 1 (line 10) and we check $v$ (line 11):

- if $v$ has never been visited (not in OPEN), we assign the split flow $curr_w$ to $v$ (line 12) and then check if $v$ is ready. If ready, we push it to the front of OPEN (line 14); otherwise, we push it to the back of OPEN (line 16).
- if $v$ is already in OPEN, we update $v$'s cumulative flow by adding the split flow $curr_w$ to it (line 18).

After processing each child $v$, we update the $f^i_{u,v}$ by adding $curr_w$ to it (line 19). Note that a CLOSED list is unnecessary in this algorithm since the graph contains no cycles; once a node has propagated its flow, it will not be revisited.

## 4.3 Guidance and Traffic Heuristics

In the previous section, we described how to identify all optimal paths for one agent and how an agent's paths contribute to the probabilistic traffic flow. In this section, we will describe how to utilise these methods to compute OCPG and traffic flow for every agent, and how a myopic MAPF planner utilises these traffic flows to compute congestion-aware movements.

To build a complete picture of system-wide traffic, we can compute the total probabilistic traffic flow, $\mathcal{F}$, in an iterative manner, similar to the process in Chen et al. [4]. We begin with an empty flow map, $\mathcal{F} = \emptyset$. Then, for each agent $a_i \in A$, we perform the following steps:

(1) We derive the current traffic costs, $traf(u, v)$, for all edges using the latest aggregated flows stored in $\mathcal{F}$.
(2) Using these costs, we compute the OCPG $G'_i$ for agent $a_i$ with Algorithm 2.
(3) We then compute the probabilistic flows $f^i_{u,v}$ and $f^i_v$ for each edge $(u, v) \in G'_i$ and vertex $v \in G'_i$ for each agent $a_i$.
(4) Finally, we update the global traffic flow by adding the agent's individual flows to $\mathcal{F}$.

This sequential process ensures that the path planning for each subsequent agent is informed by the probabilistic flow of all agents planned before it. After iterating through all agents, $\mathcal{F}$ contains the aggregated probabilistic traffic flow for the entire system.

Once all agents have contributed, the resulting flow $\mathcal{F}$ is ready for guiding the movements of agents. With the final traffic flow $\mathcal{F}$ and the traffic cost $traf(u, v)$ for every edge derived from it, we then generate a cost-to-go heuristic, $h_i(n)$, for each agent using Reverse Resumable A* (RRA*) [19] search from its goal $g_i$. This heuristic provides the optimal cost from any node $n$ to the goal, considering the traffic cost. Unlike Dijkstra [5], which computes the full heuristic table in one call, RRA* computes heuristic values lazily and on-demand. When the low-level planner requests a heuristic value for a specific node, RRA* expands its backward search only as far as necessary to determine the optimal cost for that node. The results are cached, and the state of the search is saved, allowing it to be efficiently resumed for subsequent queries. This congestion-cost aware heuristic replaces the simple distance metric in PIBT, enabling the low-level planner to make myopic decisions that are guided by a system-wide understanding of potential traffic.

## 4.4 Engineering Choices

The previous sections present the preliminaries for computing congestion-aware guidance. In this section, we describe how these methods are used in an LMAPF problem and discuss the engineering choices in this process. We present Algorithm 4 which is based on the workflow described in Section 4.3 but includes several function components that handle the task changes during the simulation, and includes an additional guidance refinement process.

At every timestep of the LMAPF simulation, we first call *FindGuidePaths(A)* to compute the OCPG for each agent, if the agent does not have one or is assigned a new goal. *ResetTrafficHeuristic($a_i$)* deletes all cached cost-to-goal heuristics in the meantime, and new heuristic values will be computed when PIBT queries them. Similar to Chen et al. [4], *RefineGuidePaths(S)* is then called to iteratively recompute the OCPG for every agent in the given subset $S$ until termination, e.g. timelimit reached. Finally, *FindNextMove(A)* computes actions for each agent using PIBT and the traffic heuristic.

In these functions, $UpdateOCPG(a_i)$ has several procedures:

(1) If the OCPG of $a_i$ already exists, remove its contribution from the traffic flow $\mathcal{F}$ and delete the OCPG.
(2) Use Algorithm 2 to compute new OCPG of $a_i$.
(3) Use the new OCPG to compute and contribute traffic flows to $\mathcal{F}$.

We then list different variants of our approach based on Algorithm 4, with different choices of parameters, that we will compare in the experiments.

*4.4.1 Probabilistic Traffic Flow Optimisation (PTFO).* In this setting, we set $S = A$ and impose no time limit on each timestep. In other words, all agents update their OCPGs and contribute traffic flows to the overall traffic flow at each timestep. We also set **Refine** = **false**, which means that $RefineGuidePaths()$ is not executed.

*4.4.2 Probabilistic Traffic Flow Optimisation: Shared Traffic Heuristic (PTFO_S).* This setting is a variant of PTFO. We let multiple agents share one Traffic Heuristic if they have the same goal location, whereas in PTFO, each agent maintains its own Traffic Heuristic. Since the Traffic Heuristic is shared, when one agent $a_i$ triggers function *ResetTrafficHeuristic($a_i$)*, all other agents share the same goal locations with $a_i$ get their cached heuristics deleted. This higher heuristic update frequency keeps the Traffic Heuristic more up-to-date.

*4.4.3 Probabilistic Traffic Flow Optimisation: Shared Traffic Heuristic and Refinement (PTFO_S_Re).* This setting is very similar to PTFO_S, but with **Refine** = **true** and **Init** = **false**. When **Refine** = **false**, the overall traffic flows $\mathcal{F}$ and OCPGs might be outdated since agents still may deviate from predictions and have a smaller OCPG as they get closer to their goal location. With (**Refine** = **true**), $RefineGuidePaths()$ is executed, allowing the OCPGs of agents to be updated and overall traffic flow to be refined within the given time limit at each timestep. We keep **Init** = **false**, so that we only keep the OCPGs and overall traffic flows up to date, but keep the frequency of traffic heuristic update unchanged.

*4.4.4 Probabilistic Traffic Flow Optimisation: Shared Traffic Heuristic and Refinement with Init (PTFO_S_Re_Init).* On top of PTFO_S_Re, we set **Init** = **true**. In this case, the Traffic Heuristic of each agent is reset more often, resulting in more frequent recomputation of the Traffic Heuristic.

**Algorithm 4** We introduce how we combine Cost-Equivalent Optimal Guide Paths and Traffic Heuristics to determine the next move for all agents $A$ at every time step. $S$ is a subset of $A$.

```
1:  function FindGuidePaths(A)
2:      for a_i ∈ A do
3:          if (a_i ∈ S ∧ OCPG(a_i) = ∅) or g_i is a new goal then
4:              ResetTrafficHeuristic(a_i)
5:              if a_i ∈ S then UpdateOCPG(a_i)

6:  function RefineGuidePaths(S)
7:      if Refine = true then
8:          while timelimit not reached do
9:              Select a_i from S
10:             UpdateOCPG(a_i)
11:             if Init = true then ResetTrafficHeuristic(a_i)

12: function FindNextMove(A)
13:     for a_i ∈ A do
14:         if π_i[t + 1] = ⊥ then
15:             PIBT(a_i, ⊥).            ▷ PIBT with Traffic Heuristic
```

*4.4.5 Probabilistic Traffic Flow Optimisation: 30% Sampling (PTFO_Sa30).* This setting is a sampling variant of PTFO. In problems with a huge number of agents, computing and maintaining OCPGs for all agents can be expensive and time-consuming. Inspired by the concept of sampling, where a subset of individuals is selected from a population to estimate characteristics of the whole population, we randomly select 30% of all agents $A$ to form the set $S$ and use their traffic flows to approximate the traffic and congestion of all agents. Agents not included in $S$ do not contribute to the flow but still rely on the Traffic Heuristics to determine their next moves at each timestep. **Refine is set to false**, so no traffic flow refinement is performed in this case.

*4.4.6 Probabilistic Traffic Flow Optimisation: 30% Sampling and Refinement (PTFO_Sa30_Re).* Compared to PTFO_Sa30, this setting sets **Refine** = **true**, allowing the traffic flow to be refined. At each timestep, agents in $S$ continue updating their OCPGs and refining the overall traffic flow whenever time permits.

## 5 EXPERIMENTS

In our experiments, we compare Probabilistic Traffic Flow Optimisation (PTFO) and its variants against two baselines: Priority Inheritance with Backtracking (PIBT) [15] and Traffic Flow Optimisation (TFO) [4]. We implement our algorithm on top of the competition start-kit[1] from the League of Robot Runners (LoRR) [2], which employs PIBT and TFO as its default planner. Our implementations[2] are written in C++ and executed on a Nectar Cloud virtual machine. Experiments are conducted on four maps: `room-64-64-8` and `ost003d`, taken from a popular benchmark set [20], as well as

[1] https://github.com/MAPF-Competition/Start-Kit
[2] The code is publicly available at https://github.com/plin0022/Start-Kit/tree/traffic_heuristics_mdd_float_no_fw_testing

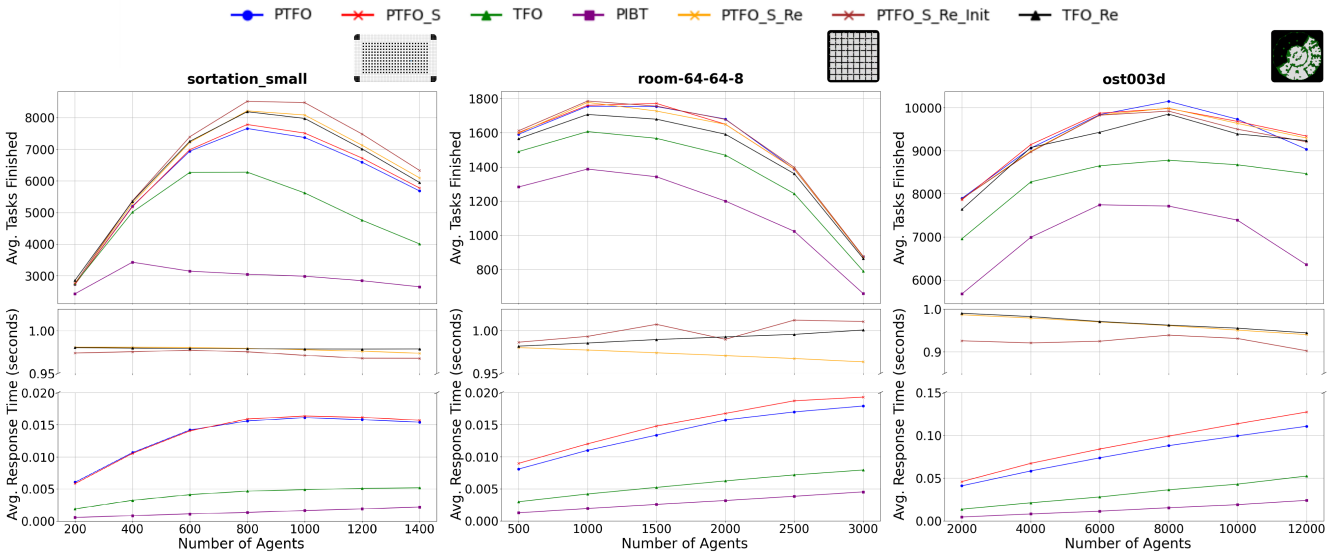



**Figure 4: Experimental results on the small and medium maps. For each map we report the average number of tasks finished (in total, higher is better) and response time (seconds, lower is better) for different numbers of agents. The response time plots in the bottom row utilise a dual-scale axis to better visualise performance variances among the (unrefined) fast approaches.**

`warehouse_large` and `sortation_small`, taken from the the LoRR competition. For the first three maps, we use 32 AMD EPYC-Rome CPUs with 64GB RAM, while for `warehouse_large`, we utilise 64 AMD EPYC-Rome CPUs with 128GB RAM.

We evaluate large-scale problem instances with up to 16,000 agents. For each map and each number of agents, we run multiple randomly sampled instances and evaluate the **Number of Tasks Finished** (or throughput) within a predefined simulation time-step limit, $T$. We also record the average time that our algorithm spent on each timestep as **Response Time**. All experiments with refinement enabled use a 1-second timelimit on each timestep[3]. The experimental configuration for each map is as follows (where minimaps are embedded at the top right of Figures 4 and 5):

**sortation_small**: A 33×57 (small) sortation centre map with 1564 traversable cells. We test 25 instances for each number of agents (200, 400, 600, 800, 1,000, 1,200, and 1,400), for a total of 175 instances. Each simulation runs for 500 timesteps.

**room-64-64-8**: A 64×64 (small) map with 3232 traversable cells. We test 25 instances for each number of agents (500, 1000, 1500, 2000, 2500, and 3000), for a total of 150 instances. Each simulation runs for 500 timesteps.

**ost003d**: A 194×194 (medium-size) map with 13214 traversable cells. We test 10 instances for each number of agents (2000, 4000, 6000, 8000, 10000, and 12000), for a total of 60 instances. Each simulation runs for 1940 timesteps.

**warehouse_large**: A 500×140 (large) warehouse fulfilment centre map with 38589 traversable cells. We test 10 instances for each number of agents (6000, 8000, 10000, 12000, 14000, and 16000), for a total of 60 instances. Each simulation runs for 3200 timesteps.

[3]There is an imperfect time limit control in the experiment results, as the number of heuristic resets affects the RRA* query time during PIBT

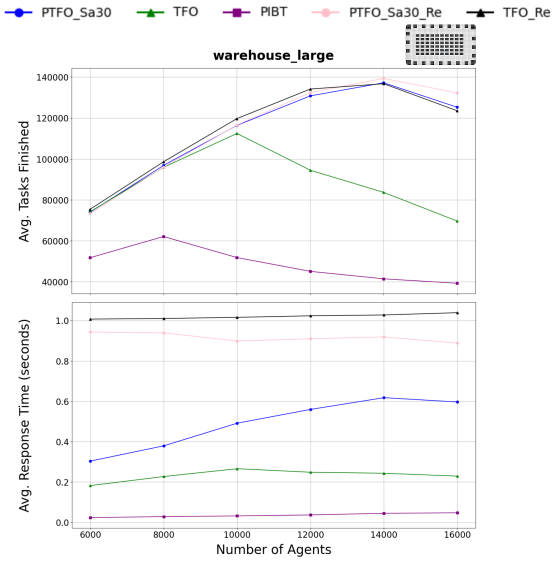



**Figure 5: Experimental results on the large map. Average tasks finished (in total, higher is better) and response time (seconds, lower is better) for different numbers of agents.**

## 5.1 Results for Small and Medium Maps

We run the experimental settings PTFO, PTFO_S, PTFO_S_Re, and PTFO_S_Re_Init (see Section 4.4 for definitions) against PIBT, TFO, and TFO_Re on `sortation_small`, `room-64-64-8`, and `ost003d` (results in Figure 4). Note that TFO_Re is TFO with refinement process enabled: TFO_Re continues refining its Guide Paths and Guide Heuristics within the 1s limit for each timestep [4].

***PTFO without Refinement.*** In Figure 4, PTFO, PTFO_S, TFO, and PIBT are algorithms without refinement. The differences between PTFO and PTFO_S are trivial. Compared to the baselines, our methods consistently achieve substantially higher throughput across all three maps. The improvement is most pronounced on the `sortation_small` map, where PTFO and PTFO_S achieve nearly double the throughput of PIBT. Even on the challenging `room-64-64-8` map, which is characterised by narrow corridors that create frequent bottlenecks, our approach maintains a considerable advantage. This trend continues on the large-scale `ost003d` map, especially as the number of agents increases. These results validate our key idea: by reasoning over the entire set of cost-equivalent paths, PTFO's single-shot estimation generates a more accurate model of traffic distribution without sacrificing the response time. This, in turn, produces a more effective congestion-aware heuristic from the outset, outperforming single-path methods even without the need for iterative refinement.

***PTFO with Refinement.*** When iterative refinement is enabled, our approach continues to demonstrate superior performance. The results in Figure 4 show that both PTFO_S_Re and PTFO_S_Re_Init consistently outperform the baseline TFO_Re across all three maps. On `sortation_small`, PTFO_S_Re_Init brings additional task completions on top of PTFO. The advantage of our model is particularly evident on the more complex `room-64-64-8` and `ost003d` maps. On these maps, even the standard PTFO without any refinement surpasses the performance of TFO_Re. This strongly demonstrates that the probabilistic traffic flow model provides a fundamentally more accurate and robust initial congestion estimate. The guidance generated by our method from a single pass is so effective that the iterative refinement process of TFO_Re cannot fully compensate for the shortcomings of its underlying single-path assumption.

Although the benefits of refinement on top of PTFO are tiny on `room-64-64-8` and `ost003d`, this suggests that the single pass probabilistic traffic flow model already provides a high-quality guidance on these types of maps, whereas TFO relies heavily on refinement for timely updates and alternative path choices. Overall, these results highlight a key advantage of our approach: PTFO-based algorithms achieve superior performance with superior response time, and refinement further improves the performance on certain types of maps.

## 5.2 Results for Large Map

We evaluate PTFO_Sa30 and PTFO_Sa30_Re against PIBT, TFO, and TFO_Re on `warehouse_large` (results in Figure 5). Since this is a large map, we only select 30% of the agents' traffic flow to approximate the overall traffic flow to reduce the amount of OCPG computation. Despite this reduced sampling, PTFO_Sa30 still demonstrates strong performance when refinement is disabled, outperforming TFO by a large margin, reaching approximately three times the throughput of PIBT when the number of agents is 14,000.

When refinement is enabled, PTFO_Sa30_Re performs slightly worse than TFO_Re at smaller scales (below 12,000 agents), but surpasses it as the number of agents increases. Similar to the first three maps, the refinement provides only marginal improvement for PTFO variants.

Additionally, PTFO_Sa30 (no refinement) delivers similar task completions compared with TFO_Re, and requires only about half the computation time of TFO_Re.

Overall, the results from this map further illustrate that PTFO-based algorithms benefit minimally from refinement within the given time limit (1s). This is in contrast to TFO, which relies heavily on replanning to maintain up-to-date congestion information.

# 6 CONCLUSIONS AND FUTURE WORKS

In this paper, we addressed a key limitation in state-of-the-art LMAPF guidance, where congestion is estimated based on the fragile assumption that each agent will follow a single, pre-determined path. We proposed Probabilistic Traffic Flow Optimisation (PTFO), a novel approach that computes more robust, congestion-aware heuristics by reasoning over the entire set of cost-equivalent optimal paths for each agent. Our extensive experiments demonstrate that PTFO significantly outperforms existing methods across a variety of maps and scales. Without iterative refinement, PTFO's single-shot guidance consistently yields higher throughput than both PIBT and the single-path TFO, validating that our probabilistic model provides a more accurate initial estimation of system-wide congestion. Furthermore, our results show that while TFO relies heavily on computationally expensive refinement to correct its initial estimates, PTFO achieves superior or comparable performance from the outset, often with significantly less computation time. The robustness of our approach was further confirmed in large-scale scenarios, where a sampling-based variant of PTFO provided high-quality guidance using only a fraction of the agents' flow data. By embracing the ambiguity of multiple optimal paths instead of ignoring it, PTFO generates more reliable and effective guidance, leading to substantial improvements in throughput for large-scale LMAPF systems.

Future work lies in three main directions. First, there are opportunities to enhance the computational performance of our implementation. For instance, the search tree generated during the OCPG computation for an agent could be cached and reused in the RRA* for the traffic heuristic, as both operate on the same cost landscape.

Second, we plan to explore more sophisticated congestion models. While the handcrafted cost function adopted in this work proved effective, recent research has shown that machine-learned congestion cost functions outperform the handcrafted one [23, 24]. Integrating our probabilistic traffic flow as a rich input feature for a learned cost function is a promising direction for further improving the accuracy of congestion prediction.

Finally, the applicability of our probabilistic guidance extends beyond myopic planners. Combining our probabilistic traffic flow model to construct guidance for more powerful planners [9, 12], could significantly improve their efficiency and solution quality.

# ACKNOWLEDGMENTS

This work is partly funded by a research gift from Amazon.

# REFERENCES

[1] Max Barer, Guni Sharon, Roni Stern, and Ariel Felner. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the Seventh Annual Symposium on Combinatorial Search*. 19–27.

[2] Shao-Hung Chan, Zhe Chen, Teng Guo, Han Zhang, Yue Zhang, Daniel Harabor, Sven Koenig, Cathy Wu, and Jingjin Yu. 2024. The league of robot runners competition: Goals, designs, and implementation. In *ICAPS 2024 System's Demonstration track*.

[3] Shao-Hung Chan, Jiaoyang Li, Graeme Gange, Daniel Harabor, Peter J Stuckey, and Sven Koenig. 2022. Flex distribution for bounded-suboptimal multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 9313–9322.

[4] Zhe Chen, Daniel Harabor, Jiaoyang Li, and Peter J Stuckey. 2024. Traffic flow optimisation for lifelong multi-agent path finding. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 20674–20682.

[5] Edsger W. Dijkstra. 2022. A Note on Two Problems in Connexion with Graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, Krzysztof R. Apt and Tony Hoare (Eds.). ACM Books, Vol. 45. ACM / Morgan & Claypool, 287–290. https://doi.org/10.1145/3544585.3544600

[6] David L Dowe, Shelton Peiris, and Eric Kim. 2024. A novel ARFIMA-ANN hybrid model for forecasting time series — and its role in explainable AI. *Journal of Econometrics and Statistics* 5, 1 (2024), 107–127. https://www.arfJournals.com/image/catalog/Journals%20Papers/JES/2025/No%201%20%282025%29/Article_7.pdf https://www.arfJournals.com/jes/issue/366.

[7] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. 2020. Sharpness-aware minimization for efficiently improving generalization. *arXiv preprint arXiv:2010.01412* (2020).

[8] Shuai D Han and Jingjin Yu. 2022. Optimizing space utilization for more effective multi-robot path planning. In *2022 International Conference on Robotics and Automation (ICRA)*. IEEE, 10709–10715.

[9] Jiang He, Yulun Zhang, Rishi Veerapaneni, and Jiaoyang Li. 2024. Scaling Lifelong Multi-Agent Path Finding to More Realistic Settings: Research Challenges and Opportunities. In *Seventeenth International Symposium on Combinatorial Search, SOCS 2024, Kananaskis, Alberta, Canada, June 6-8, 2024*, Ariel Felner and Jiaoyang Li (Eds.). AAAI Press, 234–242. https://doi.org/10.1609/SOCS.V17I1.31565

[10] Sepp Hochreiter and Jürgen Schmidhuber. 1994. Simplifying neural nets by discovering flat minima. *Advances in neural information processing systems* 7 (1994).

[11] Anthony R Jansen, David L Dowe, and Graham E Farr. 2000. Inductive inference of chess player strategy. In *Pacific Rim International Conference on Artificial Intelligence*. Springer, 61–71.

[12] He Jiang, Yutong Wang, Rishi Veerapaneni, Tanishq Duhan, Guillaume Sartoretti, and Jiaoyang Li. 2025. Deploying Ten Thousand Robots: Scalable Imitation Learning for Lifelong Multi-Agent Path Finding. In *IEEE International Conference on Robotics and Automation, ICRA 2025, Atlanta, GA, USA, May 19-23, 2025*. IEEE, 1–7. https://doi.org/10.1109/ICRA55743.2025.11127445

[13] Jiaoyang Li, Daniel Harabor, Peter J Stuckey, Hang Ma, Graeme Gange, and Sven Koenig. 2021. Pairwise symmetry reasoning for multi-agent path finding search. *Artificial Intelligence* 301 (2021), 103574.

[14] Jiaoyang Li, Andrew Tinka, Scott Kiesel, Joseph W Durham, TK Satish Kumar, and Sven Koenig. 2021. Lifelong multi-agent path finding in large-scale warehouses. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 11272–11281.

[15] Keisuke Okumura, Manao Machida, Xavier Défago, and Yasumasa Tamura. 2022. Priority inheritance with backtracking for iterative multi-agent path finding. *Artificial Intelligence* 310 (2022), 103752.

[16] Guni Sharon, Roni Stern, Ariel Felner, and Nathan R Sturtevant. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artificial Intelligence* 219 (2015), 40–66.

[17] Guni Sharon, Roni Stern, Meir Goldenberg, and Ariel Felner. 2013. The increasing cost tree search for optimal multi-agent pathfinding. *Artificial intelligence* 195 (2013), 470–495.

[18] Bojie Shen, Zhe Chen, Jiaoyang Li, Muhammad Aamir Cheema, Daniel D Harabor, and Peter J Stuckey. 2023. Beyond pairwise reasoning in multi-agent path finding. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 33. 384–392.

[19] David Silver. 2005. Cooperative pathfinding. In *Proceedings of the aaai conference on artificial intelligence and interactive digital entertainment*, Vol. 1. 117–122.

[20] Roni Stern, Nathan R Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Satish Kumar, et al. 2019. Multi-agent pathfinding: Definitions, variants, and benchmarks. In *Proceedings of the Twelfth Annual Symposium on Combinatorial Search*. 151–158.

[21] Chris S. Wallace and David L. Dowe. 1999. Minimum message length and Kolmogorov complexity. *Comput. J.* 42, 4 (1999), 270–283.

[22] Chris S Wallace and Peter R Freeman. 1987. Estimation and inference by compact coding. *Journal of the Royal Statistical Society Series B: Statistical Methodology* 49, 3 (1987), 240–252.

[23] Hongzhi Zang, Yulun Zhang, He Jiang, Zhe Chen, Daniel Harabor, Peter J. Stuckey, and Jiaoyang Li. 2025. Online Guidance Graph Optimization for Lifelong Multi-Agent Path Finding. In *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, Toby Walsh, Julie Shah, and Zico Kolter (Eds.). AAAI Press, 14726–14735. https://doi.org/10.1609/AAAI.V39I14.33614

[24] Yulun Zhang, He Jiang, Varun Bhatt, Stefanos Nikolaidis, and Jiaoyang Li. 2024. Guidance Graph Optimization for Lifelong Multi-Agent Path Finding. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI 2024, Jeju, South Korea, August 3-9, 2024*. ijcai.org, 311–320. https://www.ijcai.org/proceedings/2024/35